Reasoning About Programs

Panagiotis Manolios
Northeastern University

January 15, 2013
Version: 80

Copyright ©2012 by Panagiotis Manolios

All rights reserved. We hereby grant permission for this publication to be used for personal or classroom use. No part of this publication may be stored in a retrieval system or transmitted in any form or by any means other personal or classroom use without the prior written permission of the author. Please contact the author for details.

1

A Simple Functional Programming Language

In this chapter we introduce a simple functional programming language that forms the core of ACL2s. The language is a dialect of the Lisp programming language and is based on ACL2. In order to *reason* about programs, we first have to understand the *syntax* and *semantics* of the language we are using. The syntax of the language tells us what sequence of glyphs constitute well-formed expressions. The semantics of the language tells us what well-formed expressions (just *expressions* from now on) mean, *i.e.*, how to evaluate them. Our focus is on reasoning about programs, so the programming language we are going to use is designed to be simple, minimal, expressive, and easy to reason about.

What makes ACL2s particularly easy to reason about is the fact that it is a *functional* programming language. What this means is that every built-in function and in fact any ACL2s function a user can define satisfies the rule of Leibniz:

If $x_1 = y_1$ and $x_2 = y_2$ and $\cdots$ and $x_n = y_n$, then `(f` x_1 $x_2 \cdots x_n$`)` = `(f` y_1 $y_2 \cdots y_n$`)`

Almost no other language satisfies this very strict condition, *e.g.*, in Java you can define a function `foo` of one argument that on input 0 can return 0, or 1, or any integer because it returns the number of times it was called. This is true for Scheme, LISP, C, C++, C#, OCaml, etc. The rule of Leibniz, as we will see later, is what allows us to reason about ACL2s in a way that mimics algebraic reasoning.

You interact with ACL2s via a Read-Eval-Print-Loop (REPL). For example, ACL2s presents you with a prompt indicating that it is ready to accept input.

```
ACL2S BB !>
```

You can now type in an expression, say

```
ACL2S BB !>12
```

ACL2s reads and evaluates the expression and prints the result

```
12
```

It then presents the prompt again, indicating that it is ready for another REPL interaction

```
ACL2S BB !>
```

We recommend that as you read these notes, you also have ACL2s installed and follow along in the "Bare Bones" mode. The "`BB`" in the prompt indicates that ACL2s is in the "Bare Bones" mode.

The ACL2s programming language allows us to design programs that manipulate objects in the ACL2s *universe*. The set of all objects in the universe will be denoted by All. All includes:

- Rationals: For example, $11, -7, 3/2, -14/15$.

- Symbols: For example, `x`, `var`, `lst`, `t`, `nil`.
- Booleans: There are two Booleans, `t`, denoting *true* and `nil`, denoting *false*.
- Conses: For example, `(1)`, `(1 2 3)`, `(cons 1 ())`, `(1 (1 2) 3)`.

The Rationals, Symbols, and Conses are disjoint. The Booleans `nil` and `t` are Symbols. Conses are Lists, but there is exactly one list, the empty list, which is not a cons. We will use `()` to denote the empty list, but this is really an abbreviation for the symbol `nil`.

The ACL2s language includes a basic core of built-in functions, which we will use to define new functions.

It turns out that expressions are just a subset of the ACL2s universe. Every expression is an object in the ACL2s universe, but not conversely. As we introduce the syntax of ACL2s, we will both identify what constitutes an expression and what these expressions mean as follows. If *expr* is an expression, then

$$[\![expr]\!]$$

will denote the semantics of *expr*, or what *expr* evaluates to when submitted to ACL2s at the REPL.

We will introduce the ACL2s programming language by first introducing the syntax and semantics of constants, then Booleans, then numbers, and then conses and lists.

1.1 Constants

All constants are expressions. The ACL2s Boolean constant denoting *true* is the symbol `t` and the constant denoting *false* is the symbol `nil`. These two constants are different and they evaluate to themselves.

$$[\![\texttt{t}]\!] = \texttt{t}$$
$$[\![\texttt{nil}]\!] = \texttt{nil}$$
$$\texttt{nil} \neq \texttt{t}$$

The numeric constants include the natural numbers:

$$0, 1, 2, \ldots$$

The negative integers:

$$-1, -2, -3, \ldots$$

All integers evaluate to themselves, *e.g.*,

$$[\![3]\!] = 3$$
$$[\![-12]\!] = -12$$

And the rationals:

$$1/2,\ -1/2,\ 1/3,\ -1/3,\ 3/2,\ -3/2,\ 2/3,\ -2/3, \ldots$$

We will describe the evaluation of rationals in Section 1.3.

1.2 Booleans

There are two built-in functions, `if` and `equal`.

When we introduce functions, we specify their *signature*. The signature of `if` is:

$$\texttt{if} : \mathsf{Boolean} \times \mathsf{All} \times \mathsf{All} \rightarrow \mathsf{All}$$

The signature of `if` tells us that `if` takes three arguments, where the first argument is a Boolean and the rest of the arguments are anything at all. It returns anything. So, the signature specifies not only the *arity* of the function (how many arguments it takes) but also its input and output contracts.

Examples of `if` expressions include the following.

```
(if t nil t)
```

```
(if nil 3 4)
```

All function applications in ACL2s are written in prefix form as shown above. For example, instead of $3 + 4$, in ACL2s we write `(+ 3 4)`. The `if` expressions above are elements of the ACL2s universe, *e.g.*, the first `if` expression is a list consisting of the symbols `t`, `nil`, and `t`, in that order.

Not every list starting with the symbol `if` is an expression, *e.g.*, the following are *not* expressions.

```
(if t nil)
```

```
(if 1 3 4)
```

The first list above does not satisfy the signature of `if`, which tells us that the function has an arity of three. The second list also does not satisfy the signature of `if`, which tells us that the input contract requires that the first argument is a Boolean. In general, a list is an expression if it satisfies the signature of a built-in or previously defined function.

The semantics of `(if` *test then else*`)` is as follows.

$$[\![\texttt{(if } \textit{test } \textit{then } \textit{else}\texttt{)}]\!] = [\![\textit{then}]\!], \text{ when } [\![\textit{test}]\!] = \texttt{t}$$

$$[\![\texttt{(if } \textit{test } \textit{then } \textit{else}\texttt{)}]\!] = [\![\textit{else}]\!], \text{ when } [\![\textit{test}]\!] = \texttt{nil}$$

For all ACL2s functions we consider, we specify the semantics of the functions only in the case that the signature of the function is satisfied, *i.e.*, only for expressions. If the input contract is violated, then we say that a contract violation has occurred and the function does not evaluate to anything; hence, it does not return a result. For example, as we have seen `(if 1 3 4)` is not an expression. If you try to evaluate it you will get an error message indicating that a contract violation has occurred.

Our first function, `if`, is an important and special function. In contrast to every other function, `if` is evaluated in a *lazy* way by ACL2s. Here is how evaluation works. To evaluate

$$[\![\texttt{(if } \textit{test } \textit{then } \textit{else}\texttt{)}]\!]$$

ACL2s performs the following steps.

1. First, ACL2s evaluates *test*, *i.e.*, it computes $[\![\textit{test}]\!]$.

2. If $[\![test]\!] = \texttt{t}$, then ACL2s returns $[\![then]\!]$.

3. Otherwise, it returns $[\![else]\!]$.

Notice that *test* is always evaluated, but only one of *then* or *else* is evaluated. In contrast, for all other functions we define, ACL2s will evaluate them in a *strict* way by evaluating all of the arguments to the function and then applying the function to the evaluated results.

Examples of the evaluation of `if` expressions include the following:

$$[\![\texttt{(if t nil t)}]\!] = \texttt{nil}$$

$$[\![\texttt{(if nil 3 4)}]\!] = 4$$

Here is a more complex `if` expression.

```
(if (if t nil t) 1 2)
```

This may be confusing because it seems that the test of the `if` is a List, not a Boolean. However, notice that to evaluate an `if`, we evaluate the test first, *i.e.*:

$$[\![\texttt{(if t nil t)}]\!] = \texttt{nil}$$

Therefore,

$$[\![\texttt{(if (if t nil t) 1 2)}]\!] = [\![2]\!] = 2$$

The next function we consider is `equal`.

$$\texttt{equal} : \mathsf{All} \times \mathsf{All} \to \mathsf{Boolean}$$

$[\![\texttt{(equal x y)}]\!]$ is `t` if $[\![x]\!] = [\![y]\!]$ and `nil` otherwise.
Notice that `equal` always evaluates to `t` or `nil`.
Here are some examples.

$$[\![\texttt{(equal 3 nil)}]\!] = \texttt{nil}$$

$$[\![\texttt{(equal 0 0)}]\!] = \texttt{t}$$

$$[\![\texttt{(equal (if t nil t) nil)}]\!] = \texttt{t}$$

That's it for the built-in Booleans constants and functions.
Let us now define some utility functions.
We start with `booleanp`, whose signature is as follows.

$$\mathsf{All} \to \mathsf{Boolean}$$

The name is the concatenation of the word "`boolean`" with the symbol "`p`." The "`p`" indicates that the function is a *predicate*, a function that returns `t` or `nil`. We will use this naming convention in ACL2s (most of the time). Other Lisp dialects indicate predicates using other symbols, *e.g.*, Scheme uses "`?`" (pronounced "huh") instead of "`p`."

Here is how we define functions with contracts in ACL2s.

```
(defunc booleanp (x)
  :input-contract ...
  :output-contract ...
```

```
(if (equal x t)
    t
  (equal x nil)))
```

The contracts were deliberately elided. We will add them shortly, but first we discuss how to evaluate expressions involving `booleanp`.

How do we evaluate `(booleanp 3)`?

$[\![$`(booleanp 3)`$]\!]$

= { Semantics of `booleanp` }

$[\![$`(if (equal 3 t) t (equal 3 nil))`$]\!]$

= { Semantics of `equal`, $[\![$`(equal 3 t)`$]\!]=$ `nil`, Semantics of `if` }

$[\![$`(equal 3 nil)`$]\!]$

= { Semantics of `equal`, $[\![$`(equal 3 nil)`$]\!]=$ `nil` }

`nil`

Above we have a sequence of expressions each of which is equivalent to the next expression in the sequence for the reason given in the hint enclosed in curly braces. For example the first equality holds because we expanded the definition of `booleanp`, replacing the formal parameter `x` with the actual argument 3.

The next thing is: what is the input contract for `booleanp`?

It is `t` because there are no constraints on the input to the function. All *recognizers* will have an input contract of `t`. A recognizer is a function that given any element of the ACL2s universe recognizes whether it belongs to a particular subset. In the case of `booleanp`, the subset being recognized is the set of Booleans $\{$`t`, `nil`$\}$.

What about the output contract? Since `booleanp` is a recognizer it returns a Boolean! We express this as follows:

```
(booleanp (booleanp x))
```

So, all together we have:

```
(defunc booleanp (x)
  :input-contract t
  :output-contract (booleanp (booleanp x))
  (if (equal x t)
      t
    (equal x nil)))
```

What does the contract mean? Well, let us consider the general case. Say that function f with parameters $x_1, \ldots, x_n$ has the input contract ic and the output contract oc, then what the contract means is that for any assignment of values from the ACL2s universe to the variables $x_1, \ldots, x_n$, the following formula is always true.

ic Implies oc

Hence, the contract for `booleanp` means that for any element of the ACL2s universe, x,

`t` Implies `(booleanp (booleanp` x`))`

If we wanted to make the universal quantification and the implication explicit, we would write the following, where the domain of x is implicitly understood to be All.

$$\langle \forall x :: \texttt{t} \Rightarrow \texttt{(booleanp (booleanp } x\texttt{))} \rangle$$

Notice that by the relationship between $\Rightarrow$ (implication) and `if`, the above is equivalent to

$$\langle \forall x :: \texttt{(if t (booleanp (booleanp } x\texttt{)) t)} \rangle$$

By the semantics of `if`, we can further simplify this to

$$\langle \forall x :: \texttt{(booleanp (booleanp } x\texttt{))} \rangle$$

So, for any ACL2s element x, `booleanp` returns a `boolean`.
Let us continue with more basic definitions.

$$\texttt{and} : \mathsf{Boolean} \times \mathsf{Boolean} \to \mathsf{Boolean}$$

```
(defunc and (a b)
  :input-contract (if (booleanp a) (booleanp b) nil)
  :output-contract (booleanp (and a b))
  (if a b nil))
```

$$\texttt{implies} : \mathsf{Boolean} \times \mathsf{Boolean} \to \mathsf{Boolean}$$

```
(defunc implies (a b)
  :input-contract (and (booleanp a) (booleanp b))
  :output-contract (booleanp (implies a b))
  (if a b t))
```

$$\texttt{or} : \mathsf{Boolean} \times \mathsf{Boolean} \to \mathsf{Boolean}$$

```
(defunc or (a b)
  :input-contract (and (booleanp a) (booleanp b))
  :output-contract (booleanp (or a b))
  (if a t b))
```

How do we evaluate the above? Simple:

$[\![\texttt{(or t nil)}]\!]$
$= \{$ Definition of `or` $\}$
$[\![\texttt{(if t t nil)}]\!]$
$= \{$ Semantics of `if` $\}$
If $[\![\texttt{t}]\!] = \texttt{nil}$ then $[\![\texttt{nil}]\!]$ else $[\![\texttt{t}]\!]$
$= \{$ Constants evaluate to themselves $\}$
If $\texttt{t} = \texttt{nil}$ then `nil` else `t`
$= \{$ `t` is not `nil` $\}$
`t`

Exercise 1.1 *Define:* `not`*,* `iff`*,* `xor`*, and other Boolean functions.*

$$\texttt{not} : \mathsf{Boolean} \to \mathsf{Boolean}$$

```
(defunc not (a)
  :input-contract (booleanp a)
  :output-contract (booleanp (not a))
  (if a nil t))
```

$$\texttt{iff} : \mathsf{Boolean} \times \mathsf{Boolean} \to \mathsf{Boolean}$$

```
(defunc iff (a b)
  :input-contract (and (booleanp a) (booleanp b))
  :output-contract (booleanp (iff a b))
  (if a b (not b)))
```

$$\texttt{xor} : \mathsf{Boolean} \times \mathsf{Boolean} \to \mathsf{Boolean}$$

```
(defunc xor (a b)
  :input-contract (and (booleanp a) (booleanp b))
  :output-contract (booleanp (xor a b))
  (if a (not b) b))
```

1.3 Numbers

We have the following built-in recognizers:

$$\texttt{integerp} : \mathsf{All} \to \mathsf{Boolean}$$

$$\texttt{rationalp} : \mathsf{All} \to \mathsf{Boolean}$$

Here is what they mean.
$[\![\texttt{(integerp x)}]\!]$ is `t` iff $[\![\texttt{x}]\!]$ is an integer.
$[\![\texttt{(rationalp x)}]\!]$ is `t` iff $[\![\texttt{x}]\!]$ is a rational.
Note that integers are rationals. This is just a statement of mathematical fact.

Notice also that ACL2s includes the *real* rationals and integers, not approximations or bounded numbers, as you might find in most other languages, including C and Java.

We also have the following functions.

$$\texttt{+} : \mathsf{Rational} \times \mathsf{Rational} \to \mathsf{Rational}$$

$$\texttt{*} : \mathsf{Rational} \times \mathsf{Rational} \to \mathsf{Rational}$$

$$\texttt{<} : \mathsf{Rational} \times \mathsf{Rational} \to \mathsf{Boolean}$$

$$\texttt{unary--} : \mathsf{Rational} \to \mathsf{Rational}$$

$$\texttt{unary-/} : \mathsf{Rational} \to \mathsf{Rational}$$

Wait, what about `(unary-/ 0)`? The contract really is:

$$\texttt{unary-/} : \mathsf{Rational} \setminus \{0\} \rightarrow \mathsf{Rational}$$

How do we express this kind of thing?

```
(defunc unary-/ (a)
  :input-contract (and (rationalp a) (not (equal a 0)))
  ...)
```

The semantics of the above functions should be clear (from elementary school). Here are some examples.

$$[\![\texttt{(+ 3/2 17/6)}]\!] = \texttt{13/3}$$

$$[\![\texttt{(* 3/2 17/6)}]\!] = \texttt{17/4}$$

$$[\![\texttt{(< 3/2 17/6)}]\!] = \texttt{t}$$

$$[\![\texttt{(unary-- -2/8)}]\!] = \texttt{1/4}$$

$$[\![\texttt{(unary-/ -2/8)}]\!] = \texttt{-4}$$

Exercise 1.2 *Define subtraction on rationals* `-` *and division on rationals* `/`*. Note that the second argument to* `/` *cannot be 0.*

Let's define some more functions, starting with a recognizer for positive integers.

$$\texttt{posp} : \mathsf{All} \rightarrow \mathsf{Boolean}$$

```
(defunc posp (a)
  :input-contract t
  :output-contract (booleanp (posp a))
  (if (integerp a)
      (< 0 a)
    nil))
```

What if we tried to define `posp` as follows?

```
(defunc posp (a)
  :input-contract t
  :output-contract (booleanp (posp a))
  (and (integerp a)
       (< 0 a)))
```

Well, notice that the contract for `<` is that we give it two rationals. How do we know that `a` is rational? What we would like to do is to test that `a` is an integer first, before testing that `(< 0 a)`, but the only way to do that is to use `if`. This is another reason why `if` is special. When checking the contracts of the `then` branch of an `if`, we can assume that the `test` is *true*; when checking the contracts of an `else` branch, we can assume that the `test` is *false*. No other ACL2s function gives us this capability. If we want to collect together assumptions in order to show that contracts are satisfied, we have to use `if`.

Exercise 1.3 *Define* `natp`*, a recognizer for natural numbers.*

We also have built-in

$$\texttt{numerator} : \mathsf{Rational} \to \mathsf{Integer}$$

$$\texttt{denominator} : \mathsf{Rational} \to \mathsf{Pos}$$

$[\![\texttt{(numerator a)}]\!]$ is the numerator of the number we get after simplifying $[\![\texttt{a}]\!]$.
$[\![\texttt{(denominator a)}]\!]$ is the denominator of the number we get after simplifying $[\![\texttt{a}]\!]$.

To simplify an integer `x`, we return `x`.

To simplify a number of the form `x/y`, where `x` is an integer and `y` a natural number, we divide both `x` and `y` by the $gcd(|\texttt{x}|, \texttt{y})$ to obtain `a/b`. If $\texttt{b} = 1$, we return `a`; otherwise we return `a/b`. Note that `b` (the denominator) is always positive.

Since rational numbers can be represented in many ways, ACL2s returns the simplest representation, *e.g.*,

$$[\![2/4]\!] = 1/2$$

$$[\![4/2]\!] = 2$$

$$[\![132/765]\!] = 44/255$$

1.4 Other Atoms

Symbols and numbers are *atoms*. The ACL2s universe includes other atoms, such as strings and characters. We'll introduce them later, as needed.

1.5 Lists

The only way to create non-atomic data is to use lists.

Our first built-in function is a recognizer for *conses*.

$$\texttt{consp} : \mathsf{All} \to \mathsf{Boolean}$$

Conses are non-empty lists and are comprised of a first element and the rest of the list. Here are the functions for accessing the first and rest of a cons.

$$\texttt{first} : \mathsf{Cons} \to \mathsf{All}$$

$$\texttt{rest} : \mathsf{Cons} \to \mathsf{All}$$

We now define `listp`, a recognizer for lists, as follows.

$$\texttt{listp} : \mathsf{All} \to \mathsf{Boolean}$$

```
(defunc listp (l)
  :input-contract t
  :output-contract (booleanp (listp l))
  (if (consp l)
      (listp (rest l))
```

```
(equal l () )))
```

The last built-in function is:

$$\texttt{cons} : \mathsf{All} \times \mathsf{List} \to \mathsf{Cons}$$

The semantics of the built-in functions is given by the following rules. Notice that the second argument to `cons` can either be `()` or a cons.

$$[\![\texttt{(cons x ())}]\!] = ([\![\texttt{x}]\!])$$
$$[\![\texttt{(cons x y)}]\!] = ([\![\texttt{x}]\!] \ \texttt{...}) \text{ where } [\![y]\!] = (\ldots)$$
$$[\![\texttt{(consp x)}]\!] = \texttt{t} \text{ iff } [\![x]\!] \text{ is a cons.}$$

Notice that since `consp` is a recognizer it returns a Boolean. So, if $[\![x]\!]$ is an atom, then then $[\![\texttt{(consp x)}]\!] = \texttt{nil}$.

Here are some examples.

$$[\![\texttt{(consp 3)}]\!] = \texttt{nil}$$
$$[\![\texttt{(consp (cons nil nil))}]\!] = \texttt{t}$$
$$[\![\texttt{(consp nil)}]\!] = \texttt{nil}$$

The semantics of `first` and `rest` is given with the following rules.

$$[\![\texttt{(first x)}]\!] = \texttt{a}, \text{where } [\![\texttt{x}]\!] = \texttt{(a ...)} \text{ for some } \texttt{a}, \ \ldots$$
$$[\![\texttt{(rest x)}]\!] = (\ldots), \text{where } [\![\texttt{x}]\!] = \texttt{(a ...)} \text{ for some } \texttt{a}, \ \ldots$$

Here are some examples.

$$[\![\texttt{(first (cons (if t 3 4) (cons 1 ())))}]\!] = 3$$
$$[\![\texttt{(first (rest (cons (if t 3 4) (cons 1 ()))))}]\!] = 1$$
$$[\![\texttt{(rest (cons (if t 3 4) (cons 1 (if t nil t))))}]\!] = \texttt{(cons 1 ())}$$

If you try evaluating `(rest (cons (if t 3 4) (cons 1 (if t () t))))` at the ACL2s command prompt, here is what ACL2s reports.

```
(1)
```

Since lists are so prevalent, ACL2s includes a special way of constructing them. Here is an example.

```
(list 1)
```

is just a shorthand for `(cons 1 ())`, *e.g.*, notice that asking ACL2s to evaluate

```
(equal (LIST 1) (cons 1 ()))
```

results in `t`. What is `list` really? (By the way notice that symbols in ACL2s, such as `list`, are case-insensitive.) It is not a function. Rather, it is a *macro*. There is a lot to say about macros, but for our purposes, all we need to know is that a macro gives us abbreviation power. In general

$\texttt{(list x}_1\ \texttt{x}_2\ \cdots\ \texttt{x}_n\texttt{)}$

abbreviates (or is shorthand for)

$\texttt{(cons x}_1\ \texttt{(cons x}_2\ \cdots\ \texttt{(cons x}_n\ \texttt{nil)}\ \cdots\texttt{))}$

1.6 Contract Violations

Consider

```
(unary-/ 0)
```

If you try evaluating this, you get an error because you violated the contract of `unary-/`. When a function is called on arguments that violate the input contract, we say that the function call resulted in an *input contract violation*. If such a contract violation occurs, then the function does not return anything.

Contract checking is more subtle than this, *e.g.*, consider the following definition.

```
(defunc foo (a)
  :input-contract (integerp a)
  :output-contract (booleanp (foo a))
  (if (posp a)
      (foo (- a 1))
    (rest a)))
```

ACL2s will not admit this function unless it can prove that every function call in the body of `foo` satisfies its contract, a process we call *body contract checking* and that `foo` satisfies its contract, a process we call *contract checking*. This yields five body contract conjectures and one contract conjecture.

Exercise 1.4 *Identify all the body contract checks and contract checks that the definition of* `foo` *gives rise to. Which (if any) of these conjectures is always true? Which (if any) of these conjectures is sometimes false?*

Notice that contract checking happens even before the function is admitted. This is called "static" checking. Another option would have been to perform this check "dynamically." That is, all the contract checking above would be performed as the code is running.

1.7 Termination

All ACL2s function definitions have to terminate on all inputs that satisfy the input contract.

For example, consider the following "definition."

```
(defunc listp (a)
  :input-contract t
  :output-contract (booleanp (listp a))
  (if (consp a)
      (listp a)
    (equal a nil)))
```

ACL2s will not accept the above definition and will report that it could not prove termination.

Let's look at another example.

Define a function that given n, returns $0 + ... + n$.

Here are some possibilities:

```
;; sum-n: integer -> integer
```

```
;; Given integer n, return 0+1+2+...+n
(defunc sum-n (n)
  :input-contract (integerp n)
  :output-contract (integerp (sum-n n))
  (if (equal n 0)
      0
    (+ n (sum-n (- n 1)))))
(check= (sum-n 5) (+ 1 2 3 4 5))
(check= (sum-n 0) 0)
(check= (sum-n 3) 6)
```

Exercise 1.5 *The above function does not terminate. Why? Change only the input contract so that it does terminate. Next, change the output contract so that it gives us more information about the type of values* `sum-n` *returns.*